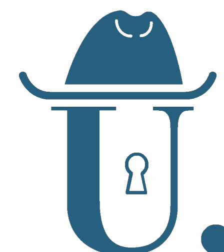
Software testing approaches

Questões de Segurança em Engenharia de Software (QSES)
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Eduardo R. B. Marques, edrdo@dcc.fc.up.pt

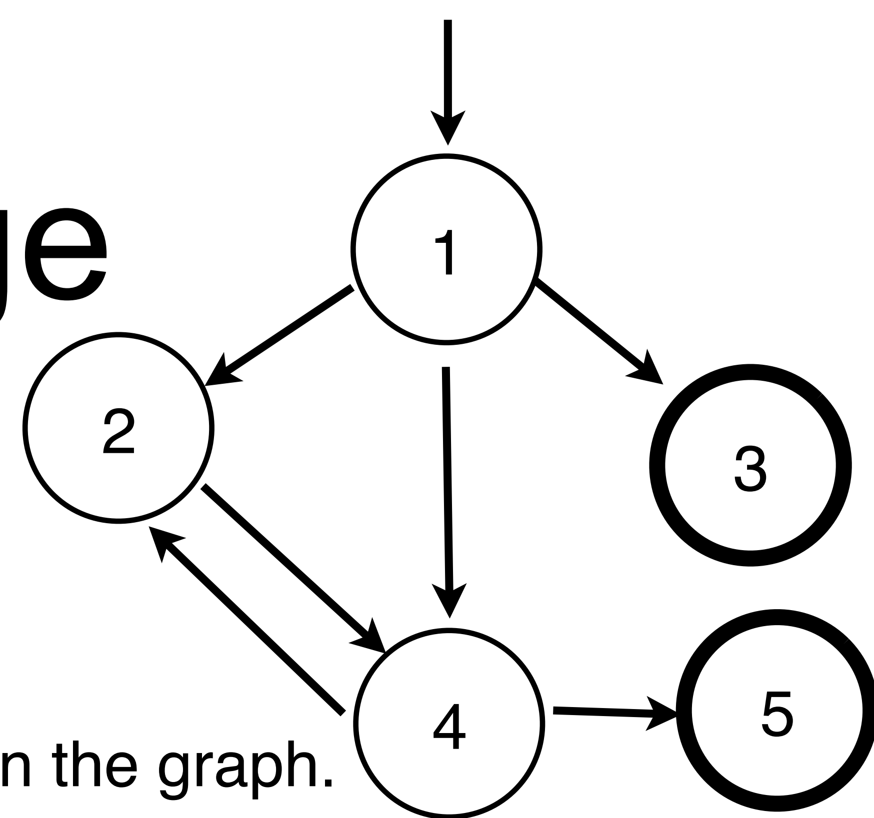


Testing approaches

- Two related questions :
  - What are meaningul inputs?
  - What (coverage) criteria should be used to derive them?
- In the last class we talked about line/instruction/branch coverage
  - Easy to understand, easy to measure through program instrumentation, and the most common metrics for coverage assessment in practice.
  - But we will also discussed how fragile they can be in transmitting a false notion regarding the quality of inputs/tests and their **ability to expose bugs!**
  - Test inputs should be derived with wits, beyond the sole purpose of maximizing line/instruction/branch coverage.
- We will briefly look at a few standard approaches
  - Graph-based coverage
  - Input space partitioning
  - Mutation testing
  - Property-based testing

Graph-based coverage

Graph-based coverage



- Basic approach
  - Model the SUT as a graph.
  - The execution of a test case corresponds to a path in the graph.
  - Coverage criteria specify requirements as sets of paths that must be covered by test paths.
- Graphs as models for:
  - individual procedures — control flow graphs (discussed next)
  - interacting units — call graphs
  - finite-state machine abstractions of software
- Structural vs. data-flow based coverage
  - Structural: takes into account only structure of the graph (example application next)
  - Data-flow based: also account for data usage in association to nodes/edges (we won't cover this)

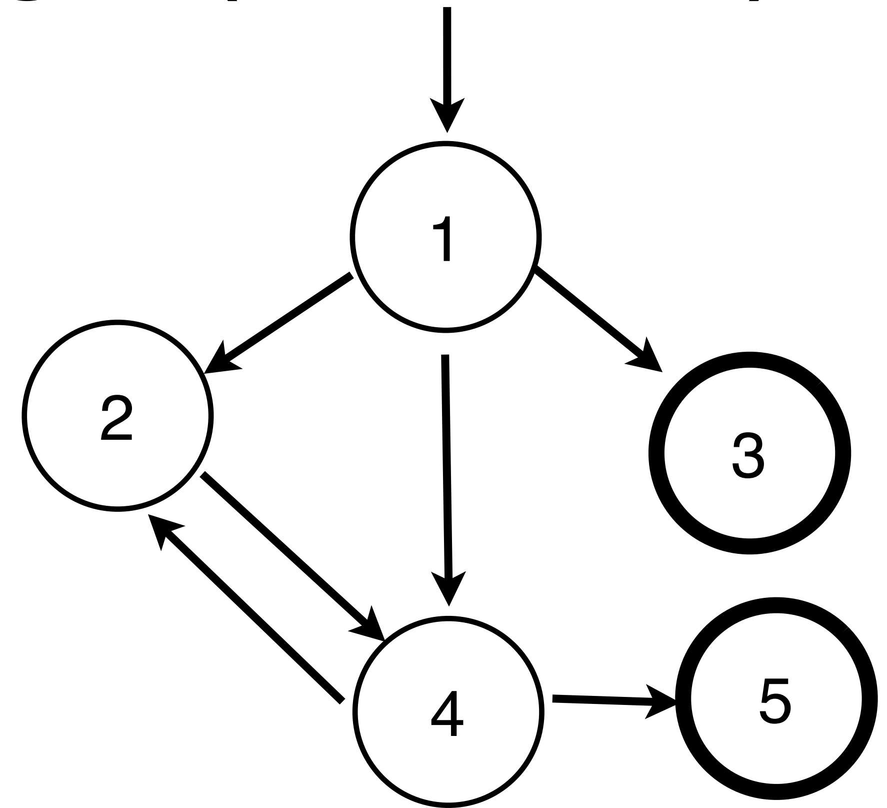
Node and edge coverage (NC, EC)

TR(NC) = { [1], [2], [3], [4], [5] }

TR(EC) = { [1,2], [1,3], [1,4], [2,4], [4,2], [4,5] }

T1 = { [1,3], [1,2,4,5] } satisfies NC, but not EC

T2 = { [1,3], [1,2,4,5],[1,4,2,4,5] }
satisfies both NC and EC



- **Node coverage (NC)**
  - Test requirements: cover every node (all graph paths up of length 0)
  - **TR(NC)** = set of nodes in the graph
- **Edge coverage (EC):** cover every edge.
  - Test requirements: cover every edge (all paths up to length 1)
  - **TR(EC)** = set of edges in the graph
- **EC subsumes NC.** Why?

Control flow graph (CFG)

- A **control flow graph (CFG)** can be used to represent the control flow of a piece of (imperative) source code.
  - **Nodes** represent **basic blocks** - sequences of instructions that always execute together in sequence.
  - **Edges** represent **control flow** between basic blocks.
  - The **entry node** corresponds to a method's entry point.
  - **Final nodes** correspond to exit points, e.g. in Java: `return` or `throw` instructions.
  - **Decision nodes** represent choices in control flow - e.g. in Java: due to `if`, `switch-case` blocks or condition tests for loops.

Example

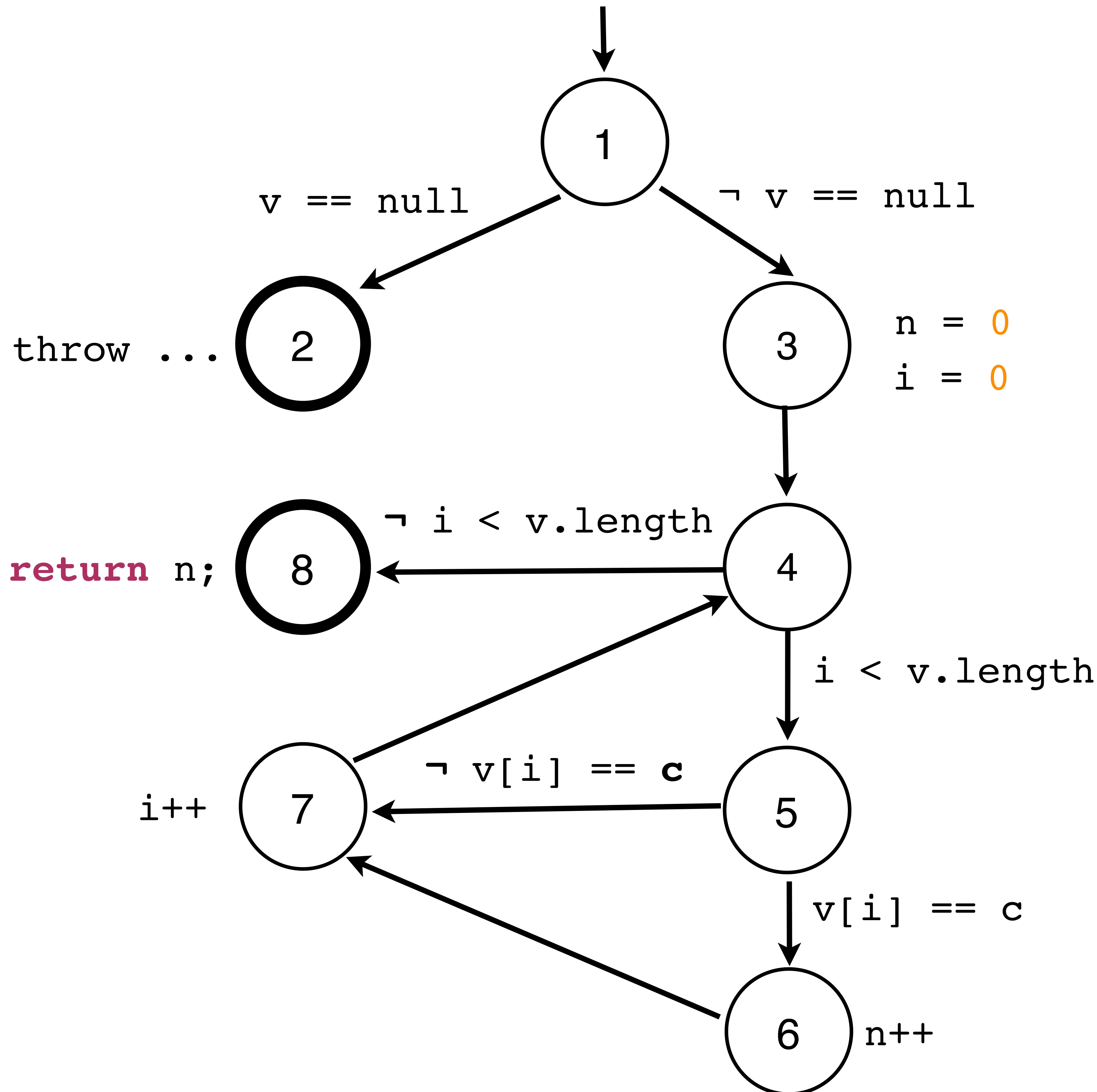
```
public static int occurrences(char[] v, char c) {
    if (v == null) {
        throw new IllegalArgumentException();
    }
    int n = 0;
    for (int i=0; i < v.length; i++) {
        if (v[i] == c) {
            n++;
        }
    }
    return n;
}
```

- Basic blocks (nodes)
  - `1: if (v == null)`
  - `2: throw ...;`
  - `3: n=0; i=0;`
  - `4: i < v.length;`
  - `5: v[i] == c;`
  - `6: n++;`
  - `7: i++;`

- Entry node
  - 1
- Decision nodes
  - 1, 4, 5
- Exit nodes
  - 2, 8

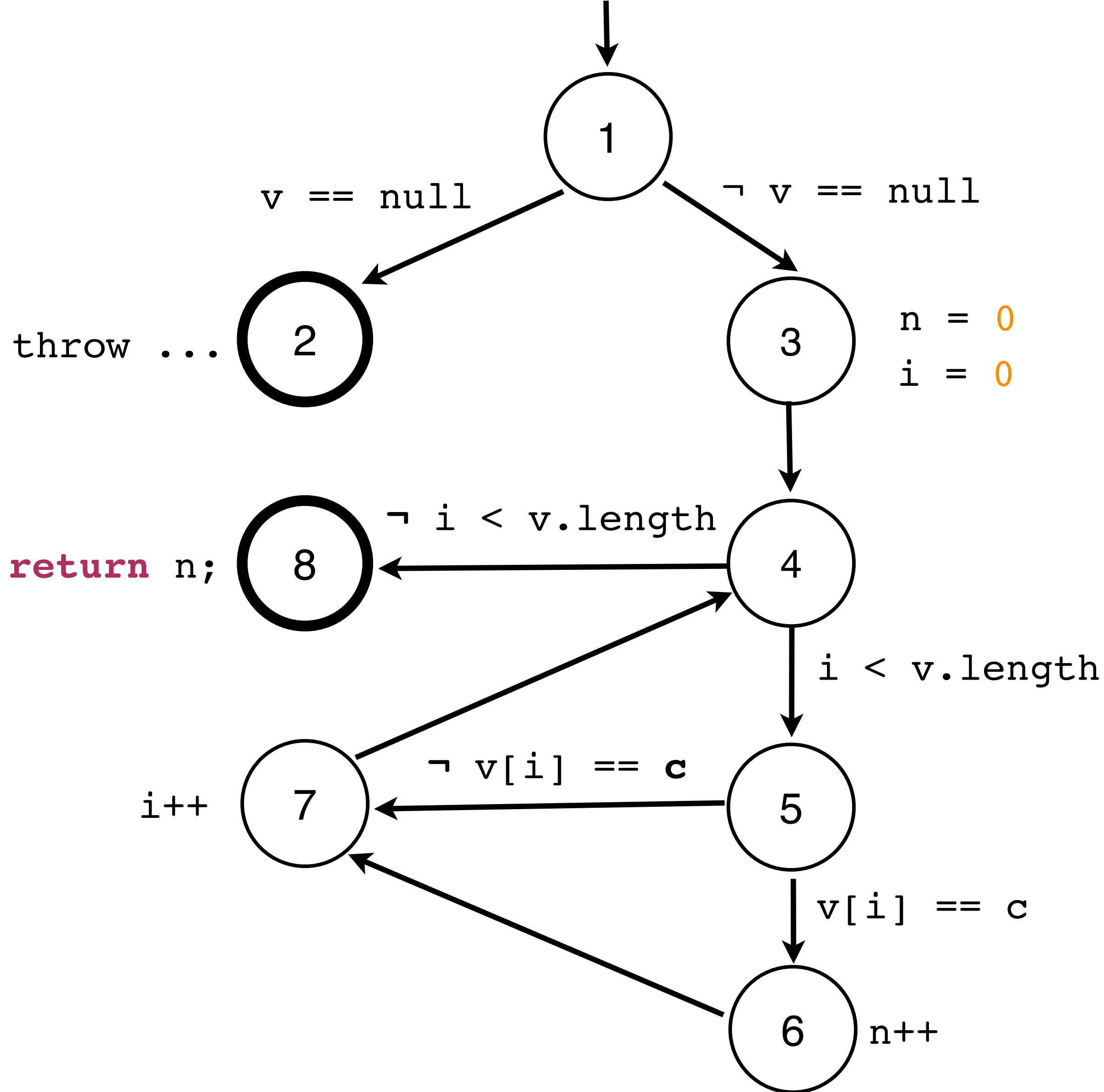
- Control flow (edges)
  - 1 → 2, 1 → 3
  - 3 → 4
  - 4 → 5, 4 → 8
  - 5 → 6, 5 → 7
  - 6 → 7
  - 7 → 4

CFG for `occurrences()`



- Basic blocks (nodes)
  - `1: if (v == null)`
  - `2: throw ...;`
  - `3: n=0; i=0;`
  - `4: i < v.length;`
  - `5: v[i] == c;`

- Control flow (edges)
  - 1 → 2, 1 → 3
  - 3 → 4
  - 4 → 5, 4 → 8
  - 5 → 6, 5 → 7
  - 6 → 7
  - 7 → 4



Node coverage

TR(NC) = { [1], [2],[3],[4], [5],[6],[7],[8] }

NC satisfied by { t1, t2 } or {t1, t3}

Edge coverage

TR(EC) = TR(NC) U {
[1,2],[1,3],[3,4],[4,5],[4,8], [5,6],[5,7][6,7],[7,4]
}

EC satisfied by { t1, t3 } but not by {t1,t2}.

t	test case values (v,c)	exp. values	test path	covered nodes	covered edges
t1	`(null, 'a')`	`IAE`	[1,2]	1 2	[1,2]
t2	`({'a'}, 'a')`	`1`	[1,3,4,5,6,7,4,8]	1 3 4 5 6 7 8	[1,3][3,4][4,5][5,6] [6,7][7,4][4,8]
t3	`({'x','a'}, 'a')`	`1`	[1,3,4,5,7,4,5,6,7,8]	1 3 4 5 6 7 8	[1,3][3,4][4,5][5,6] [6,7][7,4][5,7][4,8]

Beyond node/edge coverage

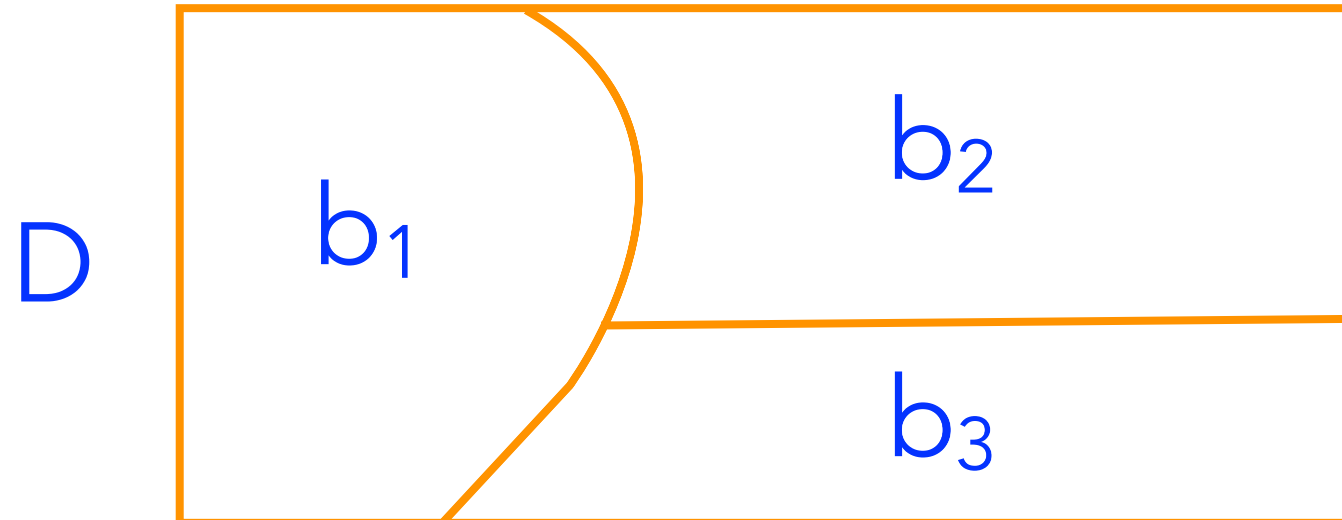
- **Edge-pair coverage (EPC)** - cover all paths up to length 2
  - **EPC** subsumes **NC** and **EPC**
- NC, EC, EPC are instances of the general criterion: cover all paths up to length k
  - NC for k=0; EC for k=1; EPC for k=2;
- As we increase k we approximate ... **Complete-Path-Coverage (CPC)**
  - **CPC:** Cover all possible paths.
  - The number of paths may be infinite or very large e.g., code with loops (CFGs with cycles) - CPC generally not applicable.
  - In practice, instead of “increasing k”, we should try to pick a subset of “relevant” paths in the graph, e.g., criteria like Prime Path Coverage [Amman & Offutt].

Input space partitioning

Input space partitioning (ISP)

- **Base idea**: identify relevant classes of input values and derive test cases from it.
- **Step 1. Identify the input parameters** for the SUT.
- **Step 2. Model the input domain** by defining one or more characteristics in the input domain. Each characteristic defines blocks that partition the input space.
- **Step 3. Apply some criterion** over characteristic of the input domain, defining a set of test requirements.
- **Step 4.** Derive test inputs (test cases).
- Also known as equivalence partitioning.

ISP - definitions



- **Input domain (D)**: the set of possible values for the input parameters.
- A **characteristic** q for **D** is a partition of D. It defines **blocks b_1 , … , b_n** such that :
  - $\forall$ **i,j : i≠j $b_i \cap b_j = \varnothing$** **(blocks are disjoint)**
  - **$D = b_1 \cup \ldots \cup b_n$** **(blocks cover the entire input domain)**
- **Q** : the set of characteristics we consider to derive test requirements.

ISP - guidelines

- **Meaningful characteristics**: Each characteristic should represent a meaningful feature for the input domain.
- **Distinctive blocks**: blocks of a characteristic should be reasonably aligned with distinctive values for it, e.g., consider:
  - “common use” values
  - boundary values
  - “invalid use” values
  - relevant relations between input parameters
- **Subdomains**: if necessary break down domain into sub-domains
  - E.g. first partition into “valid” and “invalid” values, then define characteristics for each of these domains, or sub-partition them further if convenient.

isPasswordOK example

```
/**
 * Test if password is OK.
 * @param password The password
 * @return  <code>true</code> is password is OK.
 */
boolean isPasswordOK(String password);
```

We wish to implement a password setting policy scheme, where a password is considered "OK" if all of the following conditions are met:

1. The length is between 10 and 20 characters.
2. It contains at least one upper-case character.
3. It contains at least one lower-case character.
4. It contains at least one decimal digit.
5. It may (but need not) contain at most one punctuation symbol, i.e., one of the following `. : ? ! , ;`.
6. It does not contain any character that does not fit in one of the above categories.

`isPasswordOK` example (2)

- Null vs non-null characteristic
  - Breaks domain into “null” sub-domain and “non-null” subdomain
- For the “non-null” subdomain we may consider:
  - L = Length of password
  - U = # upper-case characters
  - L = # lower-case characters
  - D = # digits
  - P = # punctuation characters
  - I = # invalid symbols
- Other meaningful characteristics ? Other relevant sub-domain characterizations?

`isPasswordOK` example (3)

- Possible blocks for the length characteristic (L)
  - L < 10, 10 <= L <= 20, L > 20 (3 blocks)
  - The blocks must define a partition. Thus, the block values do not intersect and we cannot rule out any possible value of L.
  - A more fine-grained choice could consider l=10 and l=20 blocks to force testing of boundary values for length.
- A possible choice of blocks for the X = U, L, D, and I characteristics
  - X=0, X > 0 (2 blocks each)
- Finally, for P (the punctuation characters)
  - P = 0, P = 1, P > 1

`isPasswordOK` example (4)

- Input “`Ab1234567890`” fits in the following blocks:
  - `10 <= l <= 20` (the length is 12)
  - `U > 0`
  - `L > 0`
  - `D > 0`
  - `P = 0`
  - `I = 0`
- What are the blocks for “`ABxy12!$?`” ?

ISP coverage criteria

- ***t*-wise coverage (TWC)**
  - Cover **t** blocks of different characteristics by at least one test case.
- **Each Choice Coverage (ECC) [t=1]**
  - Cover each block of each characteristic at least once.
- **Pair-wise Coverage (ECC) [t=2]**
  - Cover each block pair of two different characteristic at least once.
- **All-Combinations Coverage (ECC) [t = number of characteristics]**
  - Cover each combinations of blocks of different characteristic at least once.

ECC coverage for isPasswordOK

- A few tests are enough, for instance:
  - `"Ab1234567890"` covers blocks `10 <= l <= 20, U > 0, L > 0, D > 0, P = 0, I = 0`
  - `"!@"` covers blocks `l < 10, U = 0, L = 0, D = 0, P = 1, I > 0`
  - `"!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"` covers blocks `l > 20, U = 0, L = 0, D = 0, P > 1, I = 0`

PWC coverage for isPasswordOK

- "`Ab1234567890`" will cover 15 block pairs (5 + 4 + 3 + 2 + 1)
  - `(10 <= l <= 20, U > 0), (10 <= l <= 20, L > 0), (10 <= l <= 20, D > 0) (10 <= l <= 20, P = 0), (10 <= l <= 20, I = 0)`
  - `(U > 0, L > 0), (U > 0, D > 0), (U > 0, P = 0), (U > 0, I = 0)`
  - `(L > 0, D > 0), (L > 0, P = 0), (L > 0, I = 0),`
  - `(D > 0, P = 0), (D > 0, I = 0)`
  - `(P =0, I = 0)`
- Covering all block pairs will require more test cases.

ISP - test effort vs coverage

- **ECC**
  - $\sum_{i=1, ..., |Q|} | B_i |$ test requirements, at least $\max_{i=1, ..., |Q|} |B_i|$ tests.
  - **isPasswordOK: >= 3 tests**
- **PWC**
  - $\sum_{i,j=1, ..., |Q|, i != j} | B_i | . |B_j|$ requirements, at least M^2 tests for $M = \max_{i=1, ..., |Q|} |B_i|$
  - **isPasswordOK: ~ 3 x 3 = 9 tests**
- **ACoC**
  - $\prod_{i=1, ..., |Q|} | B_i |$ test requirements and as many tests required
  - **`isPasswordOK: 3 x 2 x 2 x 2 x 2 x 3 = 144 tests`**

Mutation testing

Mutation testing

```
public static int numZero(int[] x) {
   int count = 0;
   for (int i = 0; i < x.length; i++)
     if (x[i] == 0)
       count++;
   return count;
}
```

Introduce “faults” by mutating the code. What’s the point?

```
public static int numZero(int[] x) {
   int count = 0;
   for (int i = 1; i < x.length; i++)
     if (x[i] == 0)
       count++;
   return count;
}
```

The premise for mutation testing

- **Fundamental premise of mutation testing**
  - “*if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects the fault*” [provided we consider a rich set of mutation operators], Ammann and Offutt

sensitivity to mutations (killing mutants)

$\simeq$

sensitivity to faults (exposing failures)

"Testing the tests"

- Suppose you have a test set T for program P (maybe derived applying some coverage criteria C, manually or automatically).
- Program-based mutation testing helps answering the following key question**:**
  - ***How "good" is T (and C)?***
- For $m \in M$ (the set of all mutants), if T is "good" then a test in T should kill m.
- If no test in T kills a mutant m, then T should be reformulated (one may also question the choice of C)...
- **Program-based mutation is many times taken as the "golden standard" of coverage criteria, given its potential to subsume other testing criteria.**

Killing the mutants …

```
public static int numZero(int[] x) {
   int count = 0;
   for (int i = 1; i < x.length; i++)
     if (x[i] == 0)
       count++;
   return count;
}
```

- **i =1** is a **mutation** of **i = 0 ;** the code obtained by changing **i=0** to **i=1** is called a **mutant** of `numZero`.
- We say a test **kills the mutant** if the mutant yields different outputs from the original code.
  - Considering **`x={1,0,0}`** the mutant is **not killed**; **2** is the return value of the method for both the original code and the mutant.
  - Considering **`x={0,1,0}`** the mutant is **killed**; the result is **1** rather than **2**.

```java
public static
int min(int x, int y) {
  int v;
  if (x < y)
    v = x;
  else
    v = y;
  return v;
}
```

original code

mutants

```java
public static
int min(int x, int y) {
  int v;
  if (x >= y)
    v = x;
  else
    v = y;
  return v;
}
```

m1

```java
public static
int min(int x, int y) {
  int v;
  if (x <= y)
    v = x;
  else
    v = y;
  return v;
}
```

m2

```java
public static
int min(int x, int y) {
  int v;
  if (x < y)
    v = x;
  else
    v = -y;
  return v;
}
```

m3

Example 2

Which mutants will be killed by tests:

(t1) (x,y) = (0,0)

(t2) (x,y) = (0,1)

(t3) (x,y) = (2,1)

Observe that **m2** can not be killed. Why not?

	x	y	min	m1	m2	m3
t1	0	0	0	0	0	0
t2	0	1	0	**1**	0	0
t3	2	1	1	**2**	1	**-1**

t1 kills none of the mutants.
t2 kills m1.
t3 kills m1 and m3.

Observe that m2 will always yield the same result as the original code. Thus it cannot be killed. It is a **functionally equivalent mutant.**

mutants

m1

```
public static
int min(int x, int y) {
  int v;
  if (x >= y)
    v = x;
  else
    v = y;
  return v;
}
```

m2

```
public static
int min(int x, int y) {
  int v;
  if (x <= y)
    v = x;
  else
    v = y;
  return v;
}
```

m3

```
public static
int min(int x, int y) {
  int v;
  if (x < y)
    v = x;
  else
    v = -y;
  return v;
}
```

Mutation operators from PIT

Math Mutator (MATH)

Active by default

The math mutator replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation. The replacements will be selected according to the table below.

Original operation	Mutated operation
+	-
-	+
*	/
/	*
%	*
&	|
|	&
^	&
<<	>>
>>	<<
>>>	<<

For example

```
int a = b + c;
```

will be mutated to

```
int a = b - c;
```

http://pitest.org

Mutation operators from PIT (2)

Conditionals Boundary Mutator (CONDITIONALS_BOUNDARY)

Active by default

The conditionals boundary mutator replaces the relational operators `<, <=, >, >=`

with their boundary counterpart as per the table below.

Original conditional	Mutated conditional
<	<=
<=	<
>	>=
>=	>

For example

```
if (a < b) {
  // do something
}
```

will be mutated to

```
if (a <= b) {
  // do something
}
```

http://pitest.org

Mutation operators and effectiveness

- **Mutants to avoid …**
  - **stillborn mutant** (i.e., dead at birth): mutant is not syntactically valid
  - **functionally-equivalent mutant:** no test can kill it
  - **trivial mutant:** almost every test can kill it
- **For effectiveness, a mutation operator should:**
  - always define a syntactically valid transformation (generate no stillborn mutants)
  - generate functionally-equivalent and trivial mutants with low probability
  - mimic typical programmer mistakes
  - not be subsumed by another operator i.e., tests that kill mutants created by the other operator also kill the ones generated by this one (or a large fraction of them)

Mutation testing - coverage

- **Mutation operator o**: takes a program **P** and yields a set of mutants of **p**, **o(p)**.
- Let **O** be the set of mutation operators and **M** be the set of all mutants generated using **O** i.e., $M = \{ m \mid m \in o(p), o \in O \}$
- **Killing mutants**
  - We say a test **t** **kills** $m \in M$ iff the output of **t** for **m** differs from the output of **t** for **P**.
- **Mutation coverage** = percentage of mutants in **M** killed by at least one test.

Mutation testing tools - basics

- A MT tool has a built-in set of mutation operators. The set of mutants for the SUT is generated in automated manner according to the mutation operators.
- A test set in context is ran against the mutants. As soon as a mutant from the set is killed, it is typically not exercised by further tests.
- If the mutation coverage is not satisfactory, the test set is typically revised and/or increased with further test cases.
  - **Obs**: The strategies for both mutant generation and test selection/execution can be quite elaborate in technical terms.

Property-based testing

Property-based testing

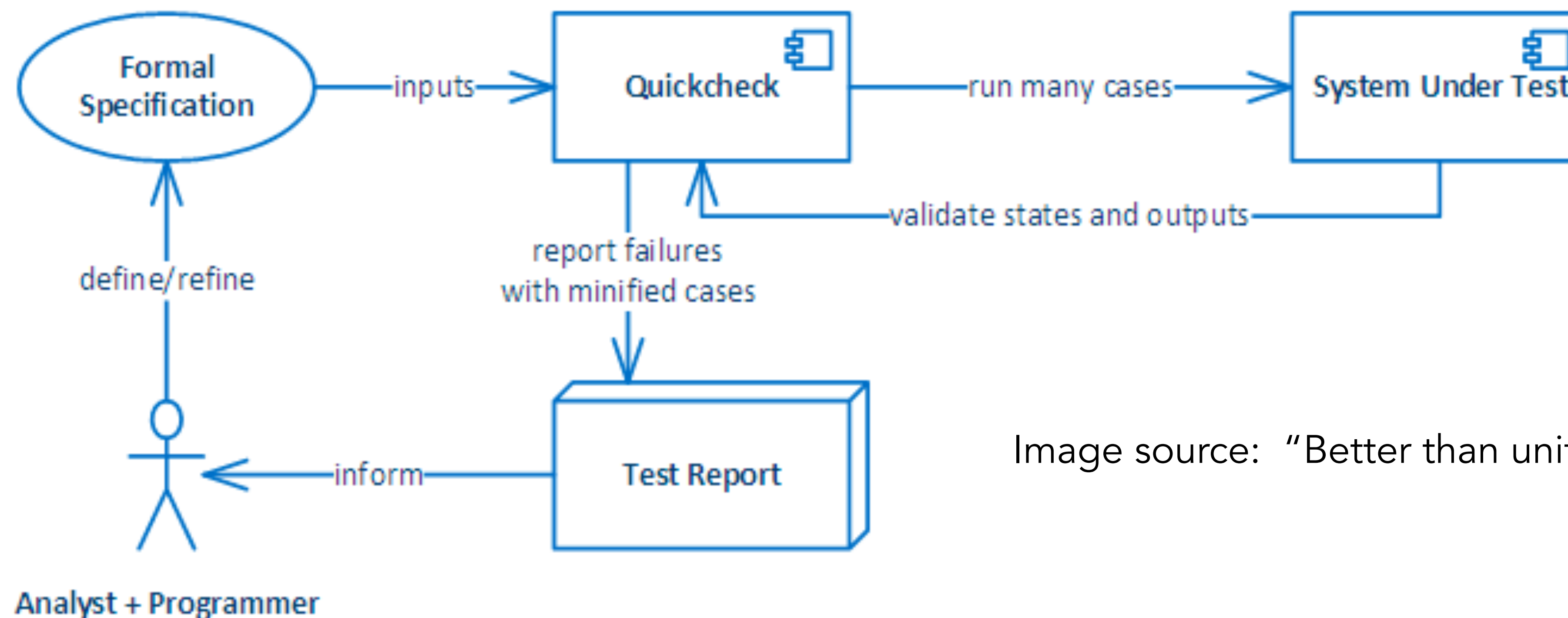


Image source: "Better than unit tests", M. Nygard

- Approach
  - Specify **properties to check** instead of inputs !
  - Let inputs be generated automatically through randomisation and customised generators.
  - If a property fails, try to find minimal input that violates the property, a process designated as **shrinking.**
- Original formulation
  - "QuickCheck: a lightweight tool for random testing of Haskell programs", Koen Claessen and John Hughes, Proc. ICFP, 2000. Adopted thereafter for other languages such as Scala or Java.

Validation of a Tiny Encryption Algorithm (TEA) implementation using QuickTheories (for Java 8)

```
@Test
 public void testTEAWithFixedKey() {
   TEA obj = new TEA("0123456789ABCDEF".getBytes());
   qt()
   .forAll(byteArrays(integers().between(1,256),
           bytes(Byte.MIN_VALUE, Byte.MAX_VALUE, (byte) 0)))
   .describedAs(data -> Arrays.toString(data))
   .check( data -> Arrays.equals(data, obj.decrypt(obj.encrypt(data))));
 }
```

fixed encryption key, but generator used for data (random byte array with length between 1 and 256)

Property: $\forall data$, decrypt(encrypt(data)) = data

```
@Test
 public void testForAnyKey() {
   Gen<Byte> anyByte = bytes(Byte.MIN_VALUE, Byte.MAX_VALUE, (byte) 0);
   Gen<byte[]> keyGen = byteArrays(constant(16), anyValue)
                         .describedAs(Arrays::toString);
   Gen<byte[]> dataGen = byteArrays(integers().between(1, 100),
anyValue).describedAs(Arrays::toString);

   qt()
   .forAll(keyGen,dataGen)
   .check( (key,data) ->  {
     TEA tea = new TEA(key);
     return Arrays.equals( tea.decrypt(tea.encrypt(data)), data);
   });
 }
```

variable key also

QuickTheories (example 2)

```
@Test
 public void testValidPasswordNoPunct() {
    Gen<Byte> lo = bytes( (byte)'a',  (byte)'z', (byte)'a');
    Gen<Byte> up = bytes( (byte)'A',  (byte) 'Z', (byte)'A');
    Gen<Byte> digit = bytes( (byte) '0', (byte) '9', (byte)'0');
    Gen<Byte> combined = lo.mix(up,50).mix(digit,25);
    Gen<byte[]> arrGen = byteArrays(integers().between(10, 20), combined);
    Gen<String> strGen = arrGen.map(ba -> new String(ba));
    qt()
      .withFixedSeed(0)
      .forAll(strGen)
      .assuming(s -> s.chars().anyMatch(Character::isLowerCase))
      .assuming(s -> s.chars().anyMatch(Character::isUpperCase))
      .assuming(s -> s.chars().anyMatch(Character::isDigit))
      .check(CHECKER::isPasswordOK);
 }
```